Beberapa Serangan pada Kriptosistem RSA: Studi Kasus pada *Challenge Capture the Flag*

{abrari, hrdn}@CySecIPB - 2016

Pendahuluan

Algoritme RSA didesain oleh Ron Rivest, Adi Shamir dan Adleman (Rivest-Shamir-Adleman) di tahun 1977, yang merupakan skema kriptografi kunci asimetris. Kekuatan algoritme RSA terletak pada faktorisasi 2 buah bilangan prima yang dibangkitkan. Berikut algoritme pembentukan kunci:

- Bangkitkan 2 bilangan prima acak p dan q yang "sangat besar", dengan selisih kedua bilangan yang juga besar.
- Ditentukan nilai $n = p \times q$ yang disebut "modulus" yang akan digunakan pada kunci privat dan kunci publik. Untuk tujuan keamanan n haruslah mempunyai nilai yang besar (misalnya 2048 bit).
- Hitung $\varphi(n)$ yang merupakan fungsi phi-euler, $\varphi(n) = \varphi(p)\varphi(q) = (p-1)(q-1)$.
- Bangkitkan nilai e yang merupakan kunci publik, digunakan untuk mengenkripsi pesan dengan persyaratan $1 < e < \varphi(n)$ dan $gcd(e, \varphi(n)) = 1$. Nilai e yang biasa digunakan adalah 65537.
- Bangkitkan nilai d yang merupakan kunci privat, digunakan untuk mendekripsi pesan dengan persyaratan $d \equiv e^{-1} \ mod \ \varphi(n)$. Dengan kata lain d merupakan *modular multiplicative inverse* dari e modulo $\varphi(n)$.

Pasangan (n, e) merupakan kunci publik yang dapat disebarluaskan dan digunakan untuk proses enkripsi. Sementara pasangan (n, d) merupakan kunci privat yang harus dijaga kerahasiaannya dan digunakan untuk proses dekripsi.

Proses Enkripsi

Setelah didapatkan kunci publik (e) dan modulus (n) pada proses pembangkitan kunci maka akan dibangkitan *ciphertext* dari *plaintext*. $C = M^e \ mod \ n$, dengan M adalah *plaintext* dan C adalah hasil enkripsi, yakni *ciphertext*.

Proses Dekripsi

Setelah didapatkan kunci privat (d) dan modulus (n) pada proses pembangkitan kunci maka akan dibangkitan *plaintext* dari *ciphertext*. $M = C^d \ mod \ n$, dengan C adalah *ciphertext* dan M adalah hasil dekripsi, yakni *plaintext*.

Kriptanalisis RSA

Seperti yang terlihat pada algoritme RSA, satu-satunya cara untuk dekripsi pesan yang terenkripsi adalah dengan mendapatkan nilai d, dengan asumsi penyerang hanya mempunyai e dan n yang didapatkan dari kunci publik dan penyerang tidak mempunyai kunci privat. Cara untuk menemukan nilai d adalah memulihkan 2 buah bilangan prima p dan q yang jika kedua bilangan tersebut dikalikan hasilnya adalah modulus (n).

Dengan kata lain, untuk mendapatkan p dan q diperlukan faktorisasi n. Belum ada algoritme yang efisien untuk memfaktorkan suatu integer yang sangat besar (misalnya ukuran n = 2048 bit) menjadi faktor-faktor primanya. Keamanan dari RSA bergantung pada hal tersebut. Jika penyerang berhasil memfaktorkan n, maka kriptosistem RSA tersebut telah berhasil diretas.

Namun demikian, jika implementasi RSA dilakukan dengan tidak tepat (misalnya pemilihan bilangan prima tidak sesuai ketentuan), akan mempermudah penyerang untuk memfaktorkan n. Pada tulisan ini akan dibahas beberapa serangan yang dapat dilakukan karena kesalahan implementasi tersebut dan studi kasusnya pada beberapa challenge kompetisi Capture the Flag (CTF). Challenge kompetisi CTF yang dibahas berasal dari CTF internasional dengan level kesulitan mudah atau sedang. Untuk teknis implementasinya, digunakan bahasa pemrograman Python dengan library *PyCrypto* dan program *OpenSSL*.

Dekripsi RSA Dasar

Pada event **PicoCTF** tahun **2014**, terdapat suatu *challenge* untuk memperkenalkan dasar dekripsi dengan algoritme RSA. Diberikan suatu file yang berisi data dari key RSA (pada contoh ini dipotong karena terlalu panjang):

```
N = 0xb197d3afe713816582e...24b661441c1ddb62ef1fe7684bbe61d8a19e7
e = 65537
p = 0xc315d99cf91a018dafb...48807060e6caec5320e3dceb25a0b98356399
q = 0xe90bbb3d4f51311f0b7...0edfaff1ad87c13eed45e1b4bd2366b49d97f
d = 0x496747c7dceae300e22...817cb53b1fb58b2a5ad163e9f1f9fe463b901
```

Dan juga *ciphertext* yang sudah dalam bentuk bilangan heksadesimal[1]:

```
58ae101736022f486216e290d39e839e7...3ed61e5ca7643ebb7fa04638aa0a0f23955e5b5d9
```

[1] RSA bekerja dalam angka, sehingga seluruh data (*plaintext* atau *ciphertext*) harus diubah terlebih dahulu dari *string* menjadi angka.

Pada *challenge* ini diberikan nilai d yang merupakan kunci privat, sehingga dapat langsung dilakukan proses dekripsi terhadap *ciphertext* dengan rumus dekripsi RSA. Berikut ini contoh implementasinya dengan bahasa Python.

```
# Data key dari soal (terpotong):
N = 0xb197d3afe713816582ee9...
e = 65537
p = 0xc315d99cf91a018dafba8...
q = 0xe90bbb3d4f51311f0b766...
d = 0x496747c7dceae300e22d5...

# Ciphertext:
C =
0x58ae101736022f486216e290d39e839e7d02a124f725865ed1b5eea7144a4c40828bd4d14dcea9675
61477a516ce338f293ca86efc72a272c332c5468ef43ed5d8062152aae9484a50051d71943cf4c3249d
8c4b2f6c39680cc75e58125359edd2544e89f54d2e5cbed06bb3ed61e5ca7643ebb7fa04638aa0a0f23
955e5b5d9

# Import fungsi yang digunakan untuk mengubah
# dari angka menjadi byte.
from Crypto.Util.number import long_to_bytes

# Lakukan pangkat C^d mod N.
M = pow(C, d, N)

# Konversi dari angka hasil pangkat ke byte.
S = long_to_bytes(M)
print(S)
```

Setelah dijalankan, didapatkan hasil dekripsinya yang merupakan jawaban untuk *challenge* ini:

```
Congratulations on decrypting an RSA message! Your flag is
modular_arithmetics_not_so_bad_after_all
```

Serangan pada RSA

Pada bagian ini akan dipaparkan lima serangan tingkat dasar yang telah diketahui pada algoritme RSA beserta contoh *challenge* CTF yang mendemonstrasikan serangan tersebut.

1. Serangan pada modulus dengan faktor prima kecil

Jika faktor prima dari modulus (p atau q) salah satunya merupakan bilangan prima yang bernilai kecil, maka n dapat difaktorkan dengan mudah meskipun nilai n besar. Salah satu caranya dengan menggunakan algoritme Pollard's rho[2]. Namun demikian, algoritme ini tidak selalu dapat memberikan hasil faktorisasi.

Pada event **Ekoparty Pre CTF 2015**, terdapat challenge dengan nama "RSA 2070". Diberikan *public key* dalam format PEM berikut:

```
-----BEGIN PUBLIC KEY-----
MIIBJDANBgkqhkiG9w0BAQEFAAOCAREAMIIBDAKCAQMlsYv184kJfRcjeGa7Uc/4
3pIkU3SevEA7CZXJfA44bUbBYcrf93xphg2uR5HCFM+Eh6qqnybpIKl3g0kGA4rv
tcMIJ9/PP8npdpVE+U4Hzf4IcgOaOmJiEWZ4smH7LWudMlOekqFTs2dWKbqzlC59
NeMPfu9avxxQ15fQzIjhvcz9GhLqb373XDcn298ueA80KK6Pek+3qJ8YSjZQMrFT
+EJehFdQ6yt6vALcFc4CB1B6qVCGO7hICngCjdYpeZRNbGM/r6ED5Nsozof1oMbt
Si8mZEJ/Vlx3gathkUVtlxx/+jlScjdM7AFV5fkRidt0LkwosDoPoRz/sDFz0qTM
5q5TAgMBAAE=
-----END PUBLIC KEY-----
```

Implementasi Python untuk membaca format PEM tersebut untuk mendapatkan n dan e terdapat pada Lampiran 1. Algoritme Pollard's rho untuk percobaan memfaktorkan n terdapat pada Lampiran 2. Berikut ini kode untuk membaca *public key* dan memfaktorkan modulusnya:

```
n, e = read_pubkey('crypto100/public.key')
print "n =", n
print "e =", e

print "Factoring using Pollard-rho..."
p = factor_pollard_rho(n)
print "p =", p
q = n / p
```

Dari hasil *running* selama beberapa detik, didapatkan salah satu faktornya yaitu $p = 3133337$ dan dapat dihitung $q = n/p$. Dengan demikian dapat dihitung juga kunci privatnya sesuai rumus algoritme RSA yang diimplementasikan pada Lampiran 3.

Ciphertext yang diberikan di-*encode* dalam Base 64 seperti berikut. Dalam *hint* dari *challenge* disebutkan tentang *padding*, maka kemungkinan *ciphertext* ini telah di-*padding* dengan PKCS1-OAEP[3].

---

[2] https://en.wikipedia.org/wiki/Pollard%27s_rho_algorithm
[3] https://en.wikipedia.org/wiki/PKCS_1

```
CQGd9sC/h9lnLpua50/071knSsP4N8WdmRsjoNIdfclrBhMjp7NoM5xy2SlNLLC2
yh7wbRw08nwjo6UF4tmGKKfcjPcb4l4bFa5uvyMY1nJBvmqQylDbiCnsODjhpB1B
JfdpU1LUKtwsCxbc7fPL/zzUdWgO+of/R9WmM+QOBPagTANbJo0mpDYxvNKRjvac
9Bw4CQTTh87moqsNRSE/Ik5tV2pkFRZfQxAZWuVePsHp0RXVitHwvKzwmN9vMqGm
57Wb2Sto64db4gLJDh9GROQN+EQh3yLoSS8NNtBrZCDddzfKHa8wv6zN/5znvBst
sDBkGyi88NzQxw9kOGjCWtwpRw==
```

Berikut ini contoh kode untuk melakukan *decode* Base 64 terhadap *ciphertext* tersebut, dan melakukan dekripsi dengan fungsi yang diimplementasikan pada Lampiran 4.

```
from base64 import b64decode
d = calculate_privkey(p, q, e)
cipher = open('crypto100/flag.enc').read()
print decrypt_oaep(n, e, d, b64decode(cipher))
```

Jika kode tersebut dijalankan, akan didapatkan *plaintext* yang merupakan jawaban dari *challenge* tersebut.

```
EKO{classic_rsa_challenge_is_boring_but_necessary}
```

2. Serangan pada modulus dengan faktor prima berdekatan

Jika faktor prima dari modulus (p dan q) nilainya berdekatan atau $|p-q|$ (selisih kedua faktor) nilainya kecil, berarti kedua faktor merupakan dua bilangan yang hampir mirip. Implikasinya, kedua faktor tersebut dapat didekati dengan akar kuadrat dari n dan kemudian dicari nilai eksaknya dengan sejumlah iterasi. Algoritme faktorisasi Fermat[4] dapat digunakan untuk mencari kedua faktor tersebut.

Pada kompetisi **Pragyan CTF 2015**, terdapat *challenge* dengan nama "Weak RSA". Diberikan suatu arsip yang mengandung file CSR dalam format PEM berikut:

```
-----BEGIN CERTIFICATE REQUEST-----
MIIBlTCB/wIBADBWMQswCQYDVQQGEwJJTjENMAsGA1UECAwETklUVDEKMAgGA1UE
BwwBLTEKMAgGA1UECgwBLTEKMAgGA1UECwwBLTEUMBIGA1UEAwwLd2Vha3JzYS5j
b20wgZ8wDQYJKoZIhvcNAQEBBQADgY0AMIGJAoGBAOiVOEnxHpMukSevNeEAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAFH4
630FVuCf//////////////////////////////////////////////////////
////////////////+61VAgMBAAGgADANBgkqhkiG9w0BAQsFAAOBgQB+2KglUaGe
e7HMBYcY/GVZutsH182VAG6u2Uutq/W1fJLowFktEd9hyNvk+Pfg0nyUmoBJIO4v
O5bwTl+rEm1tEH456cg1M3GG1g4wa0bm0ZVClNl1a88QsmL6Vkz5YAeEz3NGhSTh
N5uqhHCwPE0JRcu3UijbyjVqmyLW/IklCg==
-----END CERTIFICATE REQUEST-----
```

[4] https://en.wikipedia.org/wiki/Fermat%27s_factorization_method

Library PyCrypto tidak dapat membaca file CSR, sehingga digunakan program OpenSSL di *command line* untuk membaca informasi *public key*-nya, dan didapatkan nilai modulus yang tersimpan dalam bentuk heksadesimal.

```
$ openssl req -in domain.csr -modulus -noout

Modulus=E8953849F11E932E9127AF35E10000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000051F8EB7D0556E09FFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFBAD55
```

Modulus tersebut telah coba difaktorkan dengan algoritme Pollard's rho, namun terlalu lama waktu *running*-nya dan belum ditemukan jawaban, yang berarti kelemahannya bukan karena faktor prima yang kecil. Oleh karena itu, dicoba faktorisasi dengan algoritme Fermat. Implementasi algoritme Fermat terlampir pada Lampiran 5. Berikut ini kode percobaan faktorisasi untuk modulus tersebut:

```
n = 0xE8953849F11E932E9127AF35E10000...000051F8EB7D0556E09FFFFF...FFFFBAD55

p, q = factor_fermat(n)
print "p =", p
print "q =", q
```

Dengan cepat didapatkan kedua faktornya yang terbukti bahwa keduanya adalah bilangan prima yang mirip atau berdekatan nilanya. Dengan didapatkannya p dan q, maka dapat dikonstruksi kunci privat d dengan fungsi pada Lampiran 3.

```
p = 127798771406355522751939 74...999771218406164666202838616306272248990264 53
q = 127798771406355522751939 74...999771218406164666202838616306272248990275 21
```

Ciphertext diberikan dalam bentuk ter-*encode* Base 64, yang ketika di-*decode* merupakan angka. Dengan demikian dapat langsung dilakukan perhitungan untuk dekripsi dengan asumsi tidak ada *padding*.

```
# Hasil decode Base64 dari ciphertext.
c = 766289054801115325813389147472713847...6182132214656056698754744084973
e = 65537
d = calculate_privkey(p, q, e)
m = pow(c, d, n)

from Crypto.Util.number import long_to_bytes
print long_to_bytes(m)
```

Jika kode tersebut dijalankan, akan didapatkan *plaintext* yang merupakan jawaban dari *challenge* tersebut.

```
Congrats! The flag is too_close_primes
```

3. Serangan pada modulus dengan faktor bilangan prima Mersenne

Bilangan prima Mersenne[5] adalah bilangan prima dengan bentuk $2^m - 1$, dengan suatu nilai m tertentu. Beberapa nilai m yang valid antara lain $2, 3, 5, 7, 13, 17$. Karena bentuknya adalah dua dipangkatkan dikurangi satu, maka terdapat properti khusus dari bilangan prima ini, yaitu jika diubah menjadi bentuk biner, seluruh digitnya adalah 1. Sebagai contoh bilangan prima $2^{17} - 1 = 131071$ yang jika diubah menjadi biner menjadi 11111111111111111.

Jika suatu modulus RSA dibentuk dari dua bilangan prima Mersenne (p dan q adalah bilangan prima Mersenne), maka nilai modulusnya akan memiliki pola yang teratur. Sebagai contoh, misalnya $p = 2^{13} - 1$ dan $q = 2^{17} - 1$, maka nilai $n = p \times q = 1073602561$. Nilai n tersebut jika diubah menjadi biner adalah $111111111111011110000000000001$, yang mengandung pola satu dan nol yang berulang sejumlah pangkatnya. Jika suatu modulus mengandung pola seperti itu, dapat ditebak bahwa modulus tersebut dibentuk dari dua bilangan prima Mersenne, sehingga akan lebih mudah difaktorkan.

Contoh serangan seperti ini ada pada *challenge* dari **Backdoor CTF 2015** dengan nama "RSANNE". Diberikan suatu file *public key* dalam format PEM berikut:


```
-----BEGIN PUBLIC KEY-----
MIICUjANBgkqhkiG9w0BAQEFAAOCAj8AMIICOgKCAjEP////////////////////
////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////
//////////////////////////3////////////4AAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAECAwEAAQ==
-----END PUBLIC KEY-----
```


[5] https://en.wikipedia.org/wiki/Mersenne_prime

Modulus dari *public key* tersebut diekstrak dengan fungsi pada Lampiran 1, dan didapatkan nilai modulusnya. Jika nilai modulus tersebut diubah menjadi bentuk biner, yang panjangnya 4484 bit, terdapat pola seperti berikut:

Angka 1 sebanyak 2281 dengan satu angka 0 pada posisi 2203	Angka 0 sebanyak 2203 Dengan satu angka 1 pada posisi terakhir

Dapat "ditebak" bahwa nilai $p = 2^{2281} - 1$ dan $q = 2^{2203} - 1$. Digunakan kode program berikut untuk memastikannya, dan didapatkan jawaban *True*, yang artinya n telah berhasil difaktorkan menjadi p dan q.

```
n, e = read_pubkey('id_rsa.pub')
p = 2**2281 - 1
q = 2**2203 - 1

print p*q == n
```

Ciphertext diberikan dalam bentuk ter-*encode* Base 64, yang ketika di-*decode* menghasilkan deretan *byte* non-ASCII. Dengan demikian, dengan asumsi bahwa *ciphertext* tersebut juga di-*padding* dengan PKCS1-OAEP, digunakan kode program yang serupa dengan contoh serangan pertama (CTF Ekoparty).

```
from base64 import b64decode
d = calculate_privkey(p, q, e)
cipher = open('flag.enc').read()
print decrypt_oaep(n, e, d, b64decode(cipher))
```

Jika dijalankan, akan didapatkan *plaintext* yang merupakan jawaban dari *challenge* tersebut.

```
the_flag_is_e4972e14a8bd2430bd52d41bad8060368c2fa4f56ef6deddf1b377773a761b1a
```

4. Serangan *common factor*

Serangan *common factor*[6] dapat dilakukan jika penyerang memiliki banyak modulus, dan ada dua atau lebih dari modulus-modulus tersebut yang memiliki satu faktor prima yang sama. Misalnya terdapat dua modulus n_1 dan n_2. Modulus n_1 dibentuk dari $p \times q_1$, dan modulus n_2 dibentuk dari $p \times q_2$. Pada kedua modulus tersebut terdapat satu faktor

[6] http://www.loyalty.org/~schoen/rsa/

prima yang sama, yaitu p. Jika modulus memiliki faktor yang sama, maka penyerang dapat mencari p dengan menghitung *greatest common divisor* (GCD) dari kedua modulus, yaitu $p = gcd(n_1, n_2)$. Dengan didapatkannya p, penyerang dapat memfaktorkan n_1 dan n_2 dengan menghitung $q_1 = n_1/p$ dan $q_2 = n_2/p$, sehingga didapatkan kunci privatnya. Namun, serangan ini akan gagal jika ternyata nilai GCD dari kedua modulus adalah 1, yang berarti tidak ada faktor yang sama.

Contoh serangan *common factor* ini ada pada *challenge* dari **Backdoor CTF 2015** dengan nama "RSALOT". Diberikan 100 buah public key dalam format PEM, dan satu file *ciphertext* yang kemungkinan dienkripsi dengan salah satu dari 100 *public key* tersebut. Dengan jumlah *public key* yang banyak, kemungkinan dapat dicari faktor yang sama dengan menghitung GCD dari setiap pasang modulus, dan dicari pasangan modulus yang GCD-nya tidak sama dengan 1. Kode Python berikut mengimplementasikan pencarian faktor tersebut dengan menggunakan beberapa fungsi yang ada pada Lampiran.

```
1  import os
2  from fractions import gcd
3  from base64 import b64decode
4
5  ciphertext = open('flag.enc').read()
6  e = 65537
7
8  # Baca seluruh file dan dapatkan modulusnya.
9  moduli = []
10 for filename in os.listdir('.'):
11     if filename[-3:] == 'pem':
12        n, e = read_pubkey(filename)
13        moduli.append(n)
14
15 # Cari yang memiliki common factor (gcd != 1).
16 for i in range(len(moduli)):
17     for j in range(i+1, len(moduli)):
18         n1, n2 = moduli[i], moduli[j]
19         p = gcd(n1, n2)
20         if p != 1:
21            # Coba dekrip dengan kedua kemungkinan modulus.
22            q1 = n1 / p
23            q2 = n2 / p
24            d1 = calculate_privkey(p, q1, e)
25            d2 = calculate_privkey(p, q2, e)
26            try:
27               print decrypt_oaep(n1, e, d1, b64decode(ciphertext))
28               print decrypt_oaep(n2, e, d2, b64decode(ciphertext))
29            except:
30               pass
```

Karena faktor bersama dapat berasal dari kedua modulus (n_1 atau n_2), maka pencarian kunci privat d dan dekripsi harus dicek pada kedua modulus tersebut (kode program baris 22-28), untuk mencari modulus mana yang dapat mendekripsi *ciphertext*. Pada contoh ini modulus yang memberikan hasil dekripsi yang benar adalah n_1, sedangkan n_2 memberikan *error*. *Plaintext* hasil dekripsi dengan faktorisasi modulus n_1 adalah jawaban untuk *challenge* ini:

```
the_flag_is_b767b9d1fe02eb1825de32c6dacf4c2ef78c738ab0c498013347f4ea1e95e8fa
```

5. Serangan Wiener untuk *private key* kecil

Serangan Wiener[7] adalah serangan pada RSA yang dapat menghitung nilai d yang kecil, lebih tepatnya jika $d < \frac{1}{3}n^{1/4}$ tanpa perlu memfaktorkan n. Teorema Wiener menyebutkan bahwa hanya dengan diberikan pasangan (n, e) dengan $ed = 1 \; mod \; \varphi(n)$, penyerang dapat menghitung d dengan efisien (dengan syarat d cukup kecil seperti yang telah disebutkan). Salah satu ciri bahwa d bernilai kecil adalah jika nilai e terlalu besar (karena adanya relasi $ed = 1 \; mod \; \varphi(n)$).

Algoritme Wiener telah diimplementasikan dalam bahasa Python sebagai *library* yang dapat diunduh pada alamat https://github.com/pablocelayes/rsa-wiener-attack, yang cukup panjang sehingga tidak dilampirkan pada tulisan ini.

Salah satu *challenge* CTF yang mendemonstrasikan serangan Wiener ini adalah *challenge* dari **Volga CTF 2015** dengan nama "RSA". Diberikan suatu file *public key* dalam format PEM seperti berikut:

```
-----BEGIN PUBLIC KEY-----
MIIBHjANBgkqhkiG9w0BAQEFAAOCAQsAMIIBBgKBgDI/ranPo8MDfguQfSzqg7mt
NlUJLLBK7tlVALyk42agbLTSFcZbs9Ywt3nSe9yNzZB9ZVrL3O9GXkEb6xvj3dqr
og+wWOeFCqNV7BuJNYYC/ef4vlnUFQdwyswbd7d198qjWBZ7MiZRXxX8qKRln+os
TvsDYOMZk93k0cGZgyuJAoGAHkgFohgAnH93kDPjN4sHaT9WsmZ4ailbMtcnWuLi
zTRJ2sdGjNrpuwT1R+x1nlYHOeDUSOu6De0kQJX+HZuQCoha6THsdgcV297krN22
FwsDZ1PItXLIr5oC7zcNQaDyAJOIv6BCufHQ0IR+L9b9esniMbF8yV0d7EVAaBJi
yRk=
-----END PUBLIC KEY-----
```

Jika diekstrak nilai modulus (n) dan *exponent* (e) yang ada pada *public key* tersebut, didapatkan bahwa nilai e sangatlah besar, tidak seperti biasanya yang nilainya 65537. Hal ini memungkinkan penyerang untuk mencoba mendapatkan d dengan algoritme Wiener.

[7] https://en.wikipedia.org/wiki/Wiener%27s_attack

```
n =
352858677659170124860900583412474102996794553003407105093456832903815347985825041042606874942260249994473533997495442721289655562924974030792590184084708013341584115081540713897278190877343579417476811098257370791698230395651751021326870516013779365481737975246573155831323544237282304165774364741848753222490

e =
212642772506394736471006988462616803796694248330385714515153985222289959683936366725032014581130230428129437869854209399414978229499710386169382272395651029467414891937897538385094472349927653454871518354139227853000711635557889653618748656960811733038000349290080119559327710053022373270737250192257053350650
```

Dengan menggunakan *library* Python untuk serangan Wiener yang telah disebutkan, didapatkan nilai d dengan kode program berikut (modifikasi file *RSAwienerHacker.py*):

```
if __name__ == "__main__":
    import sys
    sys.setrecursionlimit(10000)

    n = 352858677659170124860900583412474...44237282304165774364741848753222490
    e = 21264277250639473647100698...053022373270737250192257053350650

    print "calculating d..."
    d = hack_RSA(e, n)
    print "d =", d
```

Nilai d = 374252127897518333288617847893218120810678937556096583778 1. Dapat dibuktikan bahwa $d < \frac{1}{3}n^{1/4}$ sesuai teorema Wiener. *Ciphertext* diberikan dalam format *binary* (tidak di-*encode* Base 64), sehingga dapat langsung diubah menjadi angka dan dihitung dengan rumus dekripsi RSA (dengan asumsi tidak ada *padding*). Kode program berikut melakukan dekripsi tersebut.

```
from Crypto.Util.number import long_to_bytes, bytes_to_long

d = 374252127897518333288617847893218120810678937556096583778 1
c = bytes_to_long(open('ciphertext.bin').read())
m = pow(c, d, n)
print long_to_bytes(m)
```

Didapatkan *plaintext* yang merupakan jawaban dari *challenge* ini:

```
{shorter_d_is_quicker_but_insecure}
```

Lampiran

1. Fungsi untuk memperoleh nilai n dan e dari *public key* dalam format PEM.

```python
from Crypto.PublicKey import RSA

def read_pubkey(pem_file):
    pem = open(pem_file).read()
    key = RSA.importKey(pem)
    n = key.n
    e = key.e

    return (n, e)
```

2. Algoritme Pollard's rho untuk memfaktorkan n dan mengembalikan salah satu faktor.

```python
from fractions import gcd

def factor_pollard_rho(N):
    i = 1
    power = 2
    x = y = 2
    p = 1
    while p == 1:
        i += 1
        x = (x * x + 2) % N
        p = gcd(abs(x - y), N)
        if i == power:
            y = x
            power *= 2

    if p != N: return p
    else: return None
```

3. Fungsi untuk menghitung nilai d dengan diberikan p, q dan e.

```python
from Crypto.Util.number import inverse

def calculate_privkey(p, q, e):
    phi = (p-1) * (q-1)
    d = inverse(e, phi)
    return d
```

4. Fungsi untuk melakukan dekripsi dengan *padding* PKCS1-OAEP.

```
from Crypto.PublicKey import RSA
from Crypto.Cipher import PKCS1_OAEP

def decrypt_oaep(n, e, d, ciphertext):
    rsakey = RSA.construct((n, e, d))
    rsakey = PKCS1_OAEP.new(rsakey)
    decrypted = rsakey.decrypt(ciphertext)
    return decrypted
```

5. Algoritme Fermat untuk memfaktorkan n dan mengembalikan p dan q.

```
import gmpy

def factor_fermat(N):
    a  = gmpy.sqrt(N)
    b2 = a*a - N
    while not gmpy.is_square(gmpy.mpz(b2)):
        b2 += 2*a + 1
        a  += 1

    factor1 = a - gmpy.sqrt(b2)
    factor2 = a + gmpy.sqrt(b2)
    return (long(factor1.digits()), long(factor2.digits()))
```